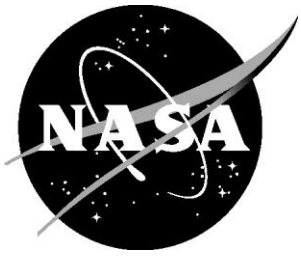
NASA/TM–2017-219655



Parallel Computation of the Jacobian Matrix for Nonlinear Equation Solvers Using MATLAB

Geoffrey K. Rose
Langley Research Center, Hampton, Virginia

Duc T. Nguyen and Brett A. Newman
Old Dominion University, Norfolk, Virginia

July 2017

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

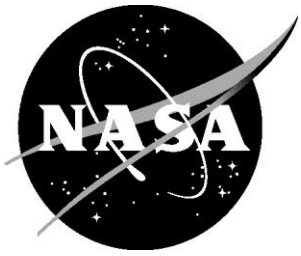
- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM–2017-219655



Parallel Computation of the Jacobian Matrix for Nonlinear Equation Solvers Using MATLAB

Geoffrey K. Rose
Langley Research Center, Hampton, Virginia

Duc T. Nguyen and Brett A. Newman
Old Dominion University, Norfolk, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

July 2017

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

Demonstrating speedup for parallel code on a multicore shared memory PC can be challenging in MATLAB due to underlying parallel operations that are often opaque to the user. This can limit potential for improvement of serial code even for the so-called embarrassingly parallel applications. One such application is the computation of the Jacobian matrix inherent to most nonlinear equation solvers. Computation of this matrix represents the primary bottleneck in nonlinear solver speed such that commercial finite element (FE) and multi-body-dynamic (MBD) codes attempt to minimize computations. A timing study using MATLAB's Parallel Computing Toolbox was performed for numerical computation of the Jacobian. Several approaches for implementing parallel code were investigated while only the single program multiple data (spmd) method using composite objects provided positive results. Parallel code speedup is demonstrated but the goal of linear speedup through the addition of processors was not achieved due to PC architecture.

Introduction

Most PCs available on the market today come equipped with multicore processors where cores share a common memory [1,2]. Programming on these systems is typically done via threading which is a special case of an operating systems process where threads share memory [2]. Multithreading or Intel's proprietary version called *hyperthreading* is also commonplace and allows for resource duplication within a given central processing unit (CPU) core [2]. Such computer architecture is what enables programming languages to exploit thread-parallel operations. Use of this technology where parallel operations are carried out autonomously without any user input or code modifications is often referred to as implicit [1] or multithreaded parallelism [3] where such operations are an integral part of the software. MATLAB software uses multithreaded parallelism by default for many of its trigonometric and linear algebraic operations [1,3]. A partial list of these functions including linear equation solvers, matrix factorization methods, etc. can be found on the MathWorks user support website [4]. This means serial versions of MATLAB code are typically running lower level parallel operations that users may be unaware of and have little or no control over. These operations can be validated in a qualitative sense through monitoring of the CPU usage history plots using Windows Task Manager or a similar program. A MATLAB serial program run using an Intel Core i7 chip for example showed use of only a single CPU for processing of a small amount of data, while processing of a significantly larger amount of data showed use of all available processors. Although the Windows Task Manager showed a total of eight available processors for this chip, it should be noted that this is a quad-core processor with eight available threads meaning four of the processors are non-physical. MATLAB still allows users to specify the number of threads being used through the *maxNumCompThreads* command [5]. Warning has however been issued by

MathWorks that this feature will be removed in a future release implying multithreading is the intended normal software environment.

Even though MATLAB exploits use of multicore processors for serial programming, code can potentially be further improved for speed through use of the Parallel Computing Toolbox. This toolbox enables use of explicit parallelism where specific tasks can be directed to specific processors. Reference to underlying parallelism for serial code could not be found in the Parallel Computing Toolbox documentation [5] while only a single reference to "built-in parallelism provided by the multithreaded nature of many of the underlying MATLAB libraries" was found in a later version. This may leave users unaware of underlying parallelism in serial code leading to high expectations for speedup of parallel versions. According to a professor who specializes in computer science, a common scenario of first-time developers of parallel code is to find out it is actually slower than the serial version which he attributes to lack of understanding of how computer hardware works, at least at a high level [2]. Establishing serial MATLAB or any computer code with underlying parallelism as the de facto standard in which to gage parallel code performance can significantly add to the challenge of achieving speedup. This can be true even for the so-called embarrassingly parallel applications as underlying parallelism may leave little room for code improvement. Users should also be aware that unlike distributed memory systems, the addition of processors for parallel computing on shared memory systems does not necessarily provide linear type improvement for speedup where doubling the number of processors doubles computational speed and so on.

MATLAB users who maintain or develop their own versions of nonlinear FE or MBD software codes may wish to speedup computations using the Parallel Computing Toolbox. For Newton-Raphson based solvers, the major cost per iteration lies in computation of the Jacobian matrix [6] where it is often referred to as the tangent stiffness matrix in FE literature. Increasing speed in which this computation is performed can have a dramatic effect on the overall solution time, especially for dynamic simulations where the matrix is not only computed during solver iterations, but at time steps during the simulation as well. One of the solver options in MBD software MSC ADAMS for example contains heuristics to help minimize the number of times computation of the Jacobian is performed as this represents the most time consuming part of a simulation [7]. Candidate algorithms for parallel computation of the Jacobian should be gaged for performance relative to a similar serial version. One of the simplest and most widely used metrics to gage parallel performance is observed speedup being defined as serial execution divided by parallel execution time in terms of total elapsed or wall-clock time [8]. This can be accomplished in MATLAB using the *tic* and *toc* functions. MATLAB also offers a function for measuring CPU time but does not recommend using it on systems capable of *hyperthreading* as the *tic* and *toc* functions are more reliable [5].

Methods for Computing the Jacobian

The Jacobian is a matrix of first-order partial derivatives resulting from the linearization or Taylor series expansion of a set of nonlinear equations about a known point or solution. This provides for a local linear model about the known point that can be used to predict nearby points in the nonlinear model. Computation of this matrix is fundamental to most nonlinear solver algorithms and is performed on an iterative basis until a converged solution to the nonlinear model is found. In commercial FE codes such as Nastran [9] and Abaqus [10], the Jacobian or tangent stiffness matrix is part of a Newton-Raphson type

solver. Due to computational expense, effective solution strategies often minimize computation or hold the Jacobian constant during iterations for a modified Newton-Raphson approach [6]. Commercial MBD software MSC ADAMS [7] also uses a Newton-Raphson type solver for dynamics and only updates the Jacobian if convergence is not achieved within a finite number of iterations. Development of efficient algorithms for computation of the Jacobian or derivatives in general is paramount to nonlinear equation solvers as this tends to dominate the total computational time for obtaining solutions.

Several methods for computing derivatives needed to construct the Jacobian are available. Review of popular FE [9,10] and MBD [7] software documentation indicates that obtaining derivatives numerically by finite difference is still the standard approach being used. A goal set by developers of MSC ADAMS is to eventually eliminate the need for numerical differentiation [11] due to high computational cost. By finite difference, derivatives of an individual function f with respect to an independent variable x are obtained by applying a small change or perturbation to x. Variable h can be used as a perturbation parameter and is added to x to represent this change. The resulting expression for the derivative of $f(x)$ or $f'(x)$ by finite difference is

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} \qquad (1)$$

which represents an approximation to the derivative by the calculus definition as it does not include the limit expression for h tending to zero. It is apparent that h cannot become too small due to limits of numerical precision on computers and possibility of dividing by a value close to zero. Take for example a sample function $f(x) = x^3 + 2x + 1$ with an exact or analytical derivative of $f'(x) = 3x^2 + 2$. Using equation (1) for estimation of the derivative about $x = 1$ and varying h by a factor of 10 between $10^0$and 10^{-20} results in figure 1 for the percent error of equation (1) with respect to the analytical derivative. Results were obtained using MATLAB with double precision representation of floating point numerical values. Error for this case was minimized for $h = 10^{-8}$ and the procedure broke down or failed for $h \leq 10^{-16}$ where $f(x)$ and $f(x + h)$ became numerically equivalent after the 15th decimal place or a maximum of 16 significant digits. The numerator in equation (1) became zero for these instances resulting in 100% error. Additional information on this method including error can be found in [12].

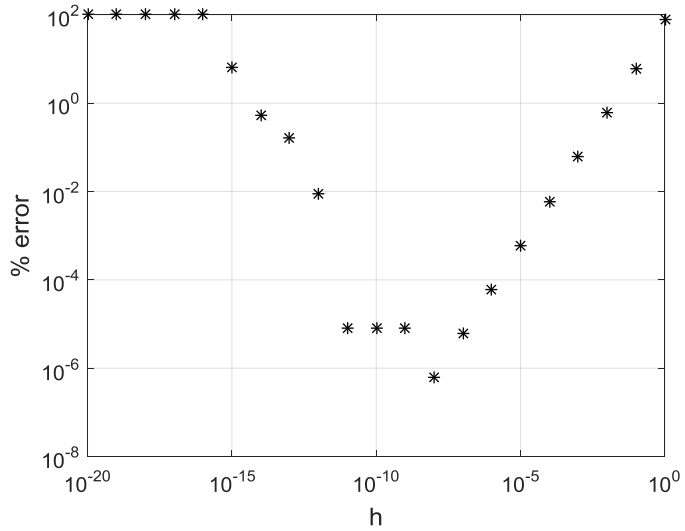


Figure 1. Percent error vs. parameter h for a given function f

An alternative to obtaining derivatives numerically by finite difference is symbolic differentiation. In this case, the symbolic expression for $f(x)$ would be differentiated using rules of calculus to obtain a new symbolic expression for $f'(x)$. Numerical values of x can then be substituted into $f'(x)$ for specific values of the derivative with an accuracy of 16 significant digits when using double precision. The result for $f'(1)$ in this case would be 5 followed by a decimal with fifteen zeros. The value obtained by finite difference on the other hand is 4.999999969612644 which exhibits error in the eighth decimal place for $h = 10^{-8}$. Although symbolic differentiation can be used to obtain derivatives in an exact sense, computational overhead for manipulating symbolic expressions using calculus based rules would limit this procedure to small problems to avoid excess solve time. A comprehensive list of computer programs capable of manipulating symbolic math expressions including their capabilities can be found at *https://en.wikipedia.org/wiki/List_of_computer_algebra_systems*.

A third alternative to obtaining derivatives is automatic differentiation. The algorithm for computing derivatives in this case uses existing computer programs or subroutines for computation of a function f and supplements them with a new routine for computation of f'. Derivatives are not subject to approximation error and are produced in an exact sense similar to the symbolic method. Automatic differentiation seems to be gaining favor based on the amount of research and computer codes being generated. Developing efficient, robust algorithms for large-scale applications has been identified as a research challenge by a developer using MATLAB [13] and favorable timing results in comparison to finite difference have been obtained for a specific class of problem by developers using C++ [14]. MSC did a study for integrating ADIFOR [15] into the FORTRAN version of ADAMS but it was not stated to having been adopted [7] implying computational overhead exceeded that of finite difference for this general purpose commercial software. A community portal with information on software, conferences, and workshops dedicated to the subject matter can be found at *http://www.autodiff.org*.

Calculation of derivatives for components of the Jacobian matrix were made using the finite difference method in both serial and parallel code versions for this study. This decision was based on ease of implementing various parallel versions for evaluating speedup and likelihood it remains the most practical approach for computing derivatives in FE and MBD programs. Equations for a repeating link or chain system were chosen for computing the Jacobian due to scalability and a specific reference in the *LSOLVER* section of the MSC ADAMS solver manual [7]. Better performance is claimed when using an available sparse matrix solver with parallel capability for systems of 5000 degrees-of-freedom (DOF) and larger with exception to some models like simply-connected long chains. This set a goal for positive margin on speedup for linkage systems under 5000 DOF for parallel computation of the Jacobian using MATLAB code. Although numerical accuracy of derivatives in the Jacobian may be of concern, highly accurate results for Newton-Raphson type solvers are not required. The modified Newton-Raphson method may hold the Jacobian constant, without any updates during iterations, and the BFGS method [16-19] which can be used as an option in both Nastran and Abaqus updates the matrix using approximations that are even less accurate than finite differences.

Equation Theory and Background

Equations for the linkage system used in this study were derived using Lagrange's method [20]. This resulted in a set of nonlinear differential algebraic equations (DAE's) used for computation of the Jacobian matrix. Equations can be represented in compact form where $\boldsymbol{u}$ is understood to contain a mix of space and time dependent variables as

$$\boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{0} \qquad (2)$$

and linearized about a known state i using a first-order Taylor series expansion for solution by the Newton-Raphson method.

$$\boldsymbol{f}(\boldsymbol{u}) \approx \boldsymbol{f}(\boldsymbol{u}_i) + \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right)_i (\boldsymbol{u} - \boldsymbol{u}_i) = \boldsymbol{0} \qquad (3)$$

Bolded terms in equation (2) and equation (3) are used to represent vectors where $\boldsymbol{u}$ is the vector of unknown variables and $\boldsymbol{f}(\boldsymbol{u})$ is the system of DAE's. Vector $\boldsymbol{u}$ is often referred to as the state vector and contains variables for position, velocity, and constraint forces for each link in the system. The derivative term in equation (3) is the Jacobian with the following expanded or matrix format for N unknown variables or DOF.

$$\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right)_i = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \cdots & \frac{\partial f_1}{\partial u_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial u_1} & \cdots & \frac{\partial f_N}{\partial u_N} \end{bmatrix}_i \qquad (4)$$

Equation (4) shows that a system containing N-DOF will have NxN or N^2 derivatives in the Jacobian. Calculation of every individual derivative may however not be required as individual equations in equation (2) can be organized in a manner such that the Jacobian will have a known pattern. This is true for mechanical systems in general and sparsity or zero-entities in the Jacobian resulting from linearization of governing DAE's can be taken advantage of as well. Details on the derivation of equations using this approach for a single link or pendulum including pattern of the Jacobian can be found in [21]. The single link has eight unknown variables for this case as motion is constrained to a plane. A similar planar constraint was used for the multi-link system in this study where total DOF is obtained by multiplying the number of links by eight. Variables or DOF for each link consist of two for position, one for orientation, their corresponding derivatives, and two for the constraint forces.

Governing equations for the multi-link systems were produced using a MATLAB function or subroutine based on a repeating pattern for systems of two links and greater. Serial and parallel subroutines with options for sparse versus dense formulations were then developed for timing of numerical computation of the Jacobian for a varying number of links. Validation of computer code was performed for a two link system under the influence of gravity using a previously developed nonlinear software suite capable of simulating dynamic systems [22]. Results for the horizontal constraint force versus time for the grounded connection with links initially configured as an upside down "V" are shown in figure 2. Blue dots on the figure were found using MATLAB and the red line was found using MSC ADAMS.

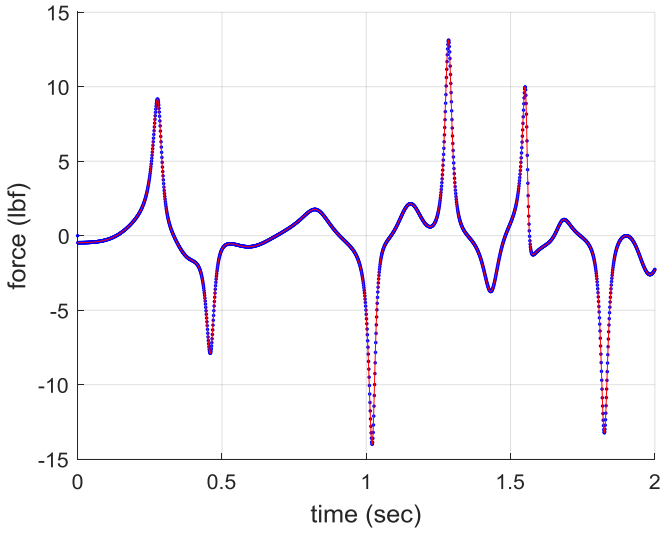


Figure 2. Constraint force vs. time for a double link system

The 16x16 Jacobian was small enough in this case where hand or symbolic computation of derivatives could be performed with reasonable effort. A function with expressions for derivative terms was then developed for computation of the Jacobian in an exact sense for comparison to a serial numerical version in terms of solution time for the two second simulation shown in figure 2. The total solution or wall time for the MATLAB simulation was 0.45 seconds using the explicit definition of the Jacobian versus a 5 second solution time for computation of derivatives numerically by finite difference. The time step used for the simulation was 0.001 seconds and convergence was achieved within 3 to 4 iterations per time step using a Newton-Raphson type solver where the Jacobian was updated at every iteration. The over tenfold increase in solution time between the two simulations demonstrates the high cost associated with numerical computation of the Jacobian. Switching to a modified Newton-Raphson method where the Jacobian was calculated numerically only once per time step and held constant increased iterations for convergence up to 9 in some instances, but reduced the solution time to 1.67 seconds. This further reinforces that computation of the Jacobian should be minimized to avoid excessive solution times in general. Note that the explicit definition of the Jacobian provided for an idealized case for timing results. Such an approach would however not be practical for large systems and require use of a numerical procedure.

Serial Code Implementation

A simplified version of MATLAB code used to numerically compute the Jacobian matrix in a serial fashion is shown in figure 3. Function *ser_jacobi* is defined to output Jacobian matrix $\boldsymbol{J}$ using state $\boldsymbol{u}_i$ as input. Code is "vectorized" in the sense that the matrix is computed a column at a time with a single for-loop verses elementwise using a double for-loop. Column entities in equation (4) show individual equations in $\boldsymbol{f}$ being differentiated with respect to a given element of vector $\boldsymbol{u}$ such that perturbations applied to specific elements of $\boldsymbol{u}$ can be used to compute entire columns of $\boldsymbol{J}$. Vectorization is a key concept in MATAB programming as it simplifies code, allows users to take advantage of underlying subroutines inherent to the programming language, and will likely perform computations in the most efficient manner. The column-wise implementation of equation (1) is shown on row twelve of figure 3. The $(:, j)$ operator is used to designate all row entities of the j^{th} column in $\boldsymbol{J}$ being a difference in perturbed vector $\boldsymbol{f}(\boldsymbol{u}_p)$ and original vector $\boldsymbol{f}(\boldsymbol{u}_i)$ with all entities being divided by h. Additional information on code vectorization can be found in the *Vectorization* section of the MATLAB user documentation [5].

```
1.    function J = ser_jacobi(u_i)
2.
3.    N    = length(u_i);              % number of elements in u
4.    fu_i = sys_eq(u_i);              % f(u) at state i
5.    h    = 1e-8;                     % perturbation parameter
6.    J    = zeros(N,N);               % pre-allocate memory for J
7.    u_p  = u_i;                      % initialize perturbation vector
8.
9.    for j = 1:N
10.       u_p(j) = u_p(j) + h;         % perturb element j in u
11.       fu_p   = sys_eq(u_p);        % f(u) at perturbed state
12.       J(:,j) = ( fu_p - fu_i )/h; % Jacobian column j derivatives
13.       u_p(j) = u_i(j);             % reset element to original value
14.   end
```

Figure 3. Serial Jacobian computation using MATLAB

Code in figure 3 is specific to computation of the full Jacobian matrix or all matrix entities and storing them in a dense format which includes any zero entities. Such computation can be expensive for large systems and a significant reduction in computational cost can be achieved by taking advantage of known patterns and sparsity. Through proper arrangement of state variables in $\boldsymbol{u}$, the Jacobian for the multi-link systems has the following block matrix format which is consistent with the general format given in [21]. Zeros sub-matrices are due to Lagrange's method being used to derive governing equations which results in large sets of equations in redundant coordinates and considerable sparsity for the Jacobian.

$$\boldsymbol{J} = \begin{bmatrix} \frac{1}{dt}\boldsymbol{M} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{\Phi}_p^T \\ \mathbf{0} & \frac{1}{dt}\boldsymbol{I_{CM}} & \mathbf{0} & [\boldsymbol{\Phi}_\varepsilon^T \boldsymbol{\Lambda}]_\varepsilon & \boldsymbol{\Phi}_\varepsilon^T \\ -\boldsymbol{I} & \mathbf{0} & \frac{1}{dt}\boldsymbol{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -\boldsymbol{I} & \mathbf{0} & \frac{1}{dt}\boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{\Phi}_p & \boldsymbol{\Phi}_\varepsilon & \mathbf{0} \end{bmatrix} \quad (5)$$

Components of $\boldsymbol{J}$ include diagonal sub-matrices $\boldsymbol{M}$, $\boldsymbol{I_{CM}}$, and $\boldsymbol{I}$ being mass, inertia, and identity matrices respectively. Term dt applied to these matrices is the time step or increment used between states for dynamic simulation. Constraint equations are stored in vector $\boldsymbol{\Phi}$ where $\boldsymbol{\Phi} = \mathbf{0}$ and subscripts p and ε are used to denote partial derivatives with respect to position and orientation variables respectively. Finally, the constraint forces or Lagrange multipliers are stored in column vector $\boldsymbol{\Lambda}$. Sub-matrices for mass, inertia and identity do not change for constant dt or within a given time step and are invariant. Standalone identity matrices are invariant by definition. This leaves only sub-matrices containing $\boldsymbol{\Phi}$ for numerical computation which dramatically reduces the amount of computational overhead and size of the for-loop in figure 3. A more efficient strategy for computation of the Jacobian would now involve pre-allocation and construction of a sparse matrix with invariant terms followed by computation of the $\boldsymbol{\Phi}$ sub-matrix blocks in the last row, and the row two, column four block locations of equation (5). Previously calculated $\boldsymbol{\Phi}$ blocks in the last row can then be transposed and inserted into the last column of equation (5).

The need for sparse versus dense format of the Jacobian is driven by both computer memory for storage and computational cost of factorization. The Jacobian must be factorized each time a new version is computed as it is part of a linear system being solved during iterations of Newton-Raphson based solvers. Eliminating the storage of zeros and the processing of zero entities in sparse computational algorithms can have dramatic effects on efficiency and become more apparent as systems increase in size. Table 1 for example shows the wall time needed to solve a sample linear system $\Delta\boldsymbol{u} = \boldsymbol{J}^{-1}\boldsymbol{R}$ where $\boldsymbol{J}$ is stored in both sparse and dense formats for timing comparison. Variable $\Delta\boldsymbol{u}$ denotes an incremental change in state vector $\boldsymbol{u}$, $\boldsymbol{R}$ is a residual vector set to all ones, and Jacobian $\boldsymbol{J}$ has been factorized into lower and upper triangular elements versus taking the inverse for solution. The speed factor in table 1 is a multiplier of how many times faster the sparse solver is compared to the dense, and the non-zero (NZ) ratio is the number of non-zero terms divided by the total or N^2 number of terms in $\boldsymbol{J}$. It is apparent from numerical values in the table that sparsity is significant and large performance gains in solution time can be expected by using the sparse matrix format and solver. A detailed overview of sparse matrices and sparse matrix operations in MATLAB can be found in [23].

Table 1 Solution times using sparse and dense Jacobian (sec)

Links	DOF	sparse	dense	factor	NZ ratio
200	1600	0.003	0.08	27.64	2.0E-03
400	3200	0.006	0.56	90.21	1.0E-03
600	4800	0.009	1.40	148.57	6.8E-04
800	6400	0.012	3.15	252.23	5.1E-04
1000	8000	0.016	5.87	372.81	4.1E-04
1200	9600	0.019	9.84	516.66	3.4E-04
1400	11200	0.023	15.06	664.97	2.9E-04
1600	12800	0.027	23.10	853.97	2.5E-04
1800	14400	0.031	32.24	1052.13	2.3E-04
2000	16000	0.034	43.29	1262.09	2.0E-04

Parallel Code Implementation

Parallel processing of computational algorithms in MATLAB can be implemented using either parallel for-loops, *parfor*, or by *spmd*. Parallel for-loops work only for the simplest of algorithms and each loop must be totally independent from all others. The perturbed vector $\boldsymbol{u}_p$ inside the for-loop shown in figure 3 is updated elementwise over the course of loop iterations such that a parallel for-loop cannot be used for computing the Jacobian in this manner. The single program multiple data or *spmd* option however is more versatile and allows for specific tasks to be assigned to specific processors. Once a parallel job is started in MATAB, one processor is assigned the role of client while the remaining processors are assigned the role of workers. Computation of the Jacobian can be accomplished by dividing the for-loop in figure 3 over a specified number of processors using *spmd*. This requires creation of an indexing array used to identify the start and finish column identification numbers based on desired matrix partitions. Changes to the serial code are however minimal making this method easy to implement.

The Jacobian can be stored using either distributed arrays, codistributed arrays or composite objects when using the *spmd* option. Arrays are considered as distributed or codistributed as viewed from the perspective of the client or worker processors. Distributed arrays are created on the client where codistributed arrays are created on the workers themselves. Positive timing results for writing and updating elements of these type arrays could not be obtained. This may be due to the client-worker relation where writing new elements to workers causes a similar update to be performed on the client. Explicit reference to how writing of elements to these arrays is performed could however not be found in documentation and users do not have access to the underlying C-code used to write MATLAB software. Composite objects on the other hand produced positive results for computing the Jacobian in parallel. These objects exist on workers, have the same variable name on all workers, but store different data. The downside of composite objects is that they must be converted back into a single matrix for use in computations in their entirety. Parallel computation of the Jacobian using composites for example will be stored in independent groups of columns on workers. If the Jacobian is then needed for use in a linear system $\boldsymbol{J}\Delta\boldsymbol{u} = \boldsymbol{R}$, it will need to be converted into matrix form.

A parallel version of the for-loop used to calculate the Jacobian in figure 3 is shown in figure 4. Code is again specific to computation of the full Jacobian matrix in dense form. This provides for the most compact, readable version of code to demonstrate *spmd* parallelization. Computation of the index array, *index*, for parsing of the Jacobian is not shown on the figure. Variable *w* is used to designate specific

worker or processor identifications. The *labindex* function is used to distribute tasks being calculation of specific columns of the Jacobian to specific workers. Column identification numbers all start with one for composite objects on workers and an additional variable *k* is used to distinguish between column identification numbers for the entire Jacobian and sections being stored in composite objects. Computation of the **Φ** blocks only would require additional indexing for start and finish row identification numbers versus processing of all rows as shown in figure 4. Final assembly of the Jacobian using dense or sparse format would then be carried out after the *spmd* block of code is complete.

```
spmd

    J = zeros(N,C);     % pre-allocate and distribute composite across workers
    u_p = u_i;          % initialize perturbation vector
    fu_i = sys_eq(u_i); % f(u) at state i

    for w = 1:NP                                    % loop over workers
        if labindex == w                            % specific worker w tasks
            for j = index(w*2-1):index(w*2)         % loop over column range in J
                u_p(j) = u_p(j) + h;                % perturb element j in u
                fu_p = sys_eq(u_p);                 % f(u) at perturbed state
                if w == 1                           % worker 1 only
                    J(:,j) = (fu_p-fu_i)/h;         % Jacobian partition derivatives
                else                                % all other workers
                    k = 2*labindex - 2;             % shift column ID to 1 for composite
                    J(:,j-index(k)) = (fu_p-fu_i)/h; % Jacobian partition derivatives
                end
                u_p(j) = u_i(j);                    % reset to original
            end
        end
    end

end
```

Figure 4. Parallel Jacobian computation using MATLAB

Code Timing Results

The timing of computer code was accomplished using the 2015b version of MATLAB software and is reported using wall time. The *tic* and *toc* functions in MATAB behave similar to a stopwatch where *toc* provides for the total elapsed or wall time since the last initiation of *tic*. The *cputime* function offers an alternative and was not used due to potential for error. Additional explanation of the two timing methods can be found in the *Measure Performance of Your Program* section of the MATLAB user documentation [5]. Here, the *tic* and *toc* functions are stated to be more reliable than *cputime* and significant difference in reported times can occur due to *hyperthreading* where instructions are processed in parallel on a single processor. Wall time may be considered a more conservative approach for characterizing performance of computer code as it includes all communications overhead associated with parallel operations.

Wall timing results for processing of the Jacobian are shown in tables 2 through 4. Each table includes a timing comparison of serial to parallel code for a given number of links using a varying number of processors (NP). The size of the Jacobian matrix or number of rows and columns is equal to the DOF number. Wall time is reported in seconds where an associated speedup factor defined as the serial divided by parallel time is used to indicate performance. Results were obtained using a Windows 7 laptop computer with an Intel i7-3720QM processor and available 16 GB RAM. A maximum of 8 threads or processors were available to MATLAB as workers and timing is initially reported using maximum

resources. This decision was based on identifying the smallest DOF system with positive performance or a speedup factor greater than one with maximum parallel communications overhead. The number of links was then varied in an increasing manor until the speedup factor no longer demonstrated significant gains in performance. At this point, use of computational resources is considered maximized with no additional bandwidth available for further performance gains. Lines across the center of tables are used to denote this breakpoint. The number of processors was then decreased while holding the DOF constant showing an expected decrease in the speedup factor due to the reduction of computational resources.

Table 2 considers computation of all entities or the full Jacobian matrix using composite objects only and saves them using dense format which includes zero terms. Positive performance with a speedup factor of 1.2 occurs for a 200 link, 1600 DOF system. As the number of DOF continues to increase, performance is seen to level off at 8000 DOF with a maximum speedup factor of 3.8. Note that linear speedup could not be obtained as the addition of processors does not come with additional memory. Decreasing the number of processors while holding DOF constant at 8000, then provides for a minimum speedup factor of 1.7 when using only two processors. Computations used for the Jacobian in table 3 were similar to those in table 2 with the exception of inclusion of time to convert the composite object to a double precision matrix. The conversion is simple but cost is significant as seen by the overall reduction in speedup factor when compared with corresponding values in table 2. Positive margin for speedup now requires a 3200 DOF versus 1600 DOF system and performance levels out at 6400 DOF versus 8000 DOF when using 8 processors.

Table 4 provides results for a pre-allocated sparse Jacobian with invariant sub-matrices and computation of the constraints or blocks containing $\mathbf{\Phi}$ only. Composite objects are used for the constraint blocks and time to convert to sparse double precision format is included as well. Gains for parallel performance are seen for systems up to 6400 DOF. Wall time is the lowest as compared to other methods and use of sparse format will provide a significant speed advantage during a linear solution phase as shown in table 1. This procedure for computing the Jacobian would be considered the most practical and recommended as it takes advantage of known patterns, sparsity, and conversion to double precision matrix format for use in solving a linear system.

Table 2 Calculation of full Jacobian, dense composite format (sec)

Links	**DOF**	**NP**	**serial**	**parallel**	**factor**
200	1600	8	0.6	0.5	1.2
400	3200	8	3.3	1.6	2.1
600	4800	8	9.7	3.1	3.1
800	6400	8	18.6	5.3	3.5
1000	8000	8	30.5	8.1	3.8
1200	9600	8	45.3	12.0	3.8
1000	8000	8	30.5	8.1	3.8
1000	8000	6	30.0	9.1	3.3
1000	8000	4	30.2	11.5	2.6
1000	8000	2	30.2	18.0	1.7

Table 3 Calculation of full Jacobian, dense matrix format (sec)

Links	DOF	NP	serial	parallel	factor
200	1600	8	0.6	0.8	0.7
400	3200	8	3.2	2.5	1.3
600	4800	8	9.9	5.3	1.9
800	6400	8	19.0	9.2	2.1
1000	8000	8	32.3	15.5	2.1
1000	8000	8	32.3	15.5	2.1
1000	8000	6	32.3	16.3	2.0
1000	8000	4	32.2	17.7	1.8
1000	8000	2	32.4	24.8	1.3

Table 4 Calculation of block Jacobian, sparse matrix format (sec)

Links	DOF	NP	serial	parallel	factor
200	1600	8	0.2	0.4	0.6
400	3200	8	1.3	0.8	1.6
600	4800	8	3.9	1.6	2.4
800	6400	8	7.8	2.7	2.9
1000	8000	8	11.8	4.0	2.9
1000	8000	8	11.8	4.0	2.9
1000	8000	6	11.3	4.9	2.3
1000	8000	4	11.3	5.9	1.9
1000	8000	2	11.4	8.4	1.4

Summary

Successful development of explicitly defined parallel code for computing the Jacobian matrix was completed using MATLAB. The *spmd* method using composite objects was found to be the only procedure that produced positive results while use of the sparse as compared to dense format provided for dramatic speed improvements for solutions to linear systems. Speedup of parallel code was demonstrated on a shared memory PC and compared to serial code with underlying parallel operations using wall time. This provided for a most conservative estimate for parallel code speedup as underlying parallel operations are integral to MATLAB and wall time includes parallel communications overhead. Linear type parallel speedup could not be achieved using the chosen performance metrics and computer architecture which are quite common and may represent a typical MATLAB environment. Performance gains were however demonstrated and an approximate three times speedup for the recommended sparse format, double precision Jacobian matrix was achieved. The goal of demonstrating speedup for systems under 5000 DOF was also achieved being most applicable to smaller scale FE or MBD problems that can run efficiently on PCs. MATLAB users running nonlinear FE and MBD codes on PCs should expect significant performance gains when using sparse matrix operations and marginal parallel performance gains for systems on the order of 3200 DOF and greater.

Acknowledgements

This work was funded by the Advanced Degree Program at NASA Langley Research Center.

References

[1] Boston University Information Services & Technology, 2017. MATLAB Parallel Computing Toolbox Tutorial. http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/, (accessed 02.06.17).

[2] N. Matloff, Programming on Parallel Machines, University of California, Davis, 2016. http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf, (accessed 02.06.17).

[3] C. Moler, Parallel MATLAB: Multiple processors and multiple cores, MathWorks (2007). https://www.mathworks.com/company/newsletters/articles/parallel-matlab-multiple-processors-and-multiple-cores.html, (accessed 02.06.17).

[4] MathWorks, 2013. MATLAB Answers. http://www.mathworks.com/matlabcentral/answers/95958-which-matlab-functions-benefit-from-multithreaded-computation, (accessed 02.06.17).

[5] MATLAB R2015b Documentation, The MathWorks, Inc., 2015.

[6] K. J. Bathe, Finite Element Procedures, Klaus-Jürgen Bathe, 2006.

[7] About ADAMS/Solver, C++ Statements, MSC Software Corporation, 2015.

[8] B. Barney, Introduction to Parallel Computing, Lawrence Livermore National Laboratory, 2016. https://computing.llnl.gov/tutorials/parallel_comp/, (accessed 02.06.17).

[9] Nastran Nonlinear User's Guide, SOL 400, MSC Software Corporation, 2016.

[10] Abaqus Analysis Users Guide, Dassault Systèmes Simulia Corp., 2014.

[11] J. Ortiz, Introduction to Adams/Solver C++, Charts from the 2011 Adams user meeting, Munich, Germany, 2011.

[12] W. Press, S. Teukolsky, W. Vetterling, B. Flannery, Numerical Recipes in Fortran 77: The Art of Scientific Computing, second ed., Press Syndicate of the University of Cambridge, 1997.

[13] R. Neidinger, Introduction to automatic differentiation and MATLAB object-oriented programming, SIAM Review, Vol. 52, No. 3, (2010) 545–563.

[14] R. Bartlett, D. Gay, E. Phipps, Automatic differentiation of C++ codes for large-scale scientific computing, in: International Conference on Computational Science, 2006, pp. 525-532.

[15] ADIFOR. http://www.anl.gov/technology/project/adifor-automatic-differentiation-fortran-77, (accessed 02.06.17).

[16] C. G. Broyden, The convergence of a class of double-rank minimization algorithms, J. Inst. Math. Appl. 6 (1970) 76–90.

[17] R. Fletcher, A new approach to variable metric algorithms, Comp. J. 13 (1970) 317–322.

[18] D. Goldfarb, A family of variable metric updates derived by variational means, Math. Comp. 24 (1970) 23–26.

[19] D. F. Shanno, Conditioning of quasi-Newton methods for function minimization, Math. Comp. 24 (1970) 647–656.

[20] L. Meirovitch, Methods of Analytical Dynamics, Dover Publications Inc., 2003.

[21] D. Negrut, A. Dyer, ADAMS/Solver Primer, MSC Software Corporation, 2004.

[22] G. Rose, D. Nguyen, B. Newman, Implementing an arc-length method for a robust approach in solving systems of nonlinear equations, in: IEEE Southeast Conference, 2016.

[23] J. Gilbert, C. Moler, R. Schreiber, Sparse matrices in MATLAB: Design and implementation, J. Matrix Anal. Appl. 13 (1992) 333-356.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-07-2017	Technical Memorandum	

4. TITLE AND SUBTITLE

Parallel Computation of the Jacobian Matrix for Nonlinear Equation Solvers Using MATLAB

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Rose, Geoffrey, K.; Nguyen, Duc T.; Newman, Brett A.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

845953.01.03.04

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

L-20845

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)

NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

NASA-TM-2017-219655

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified
Subject Category 61
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Demonstrating speedup for parallel code on a multicore shared memory PC can be challenging in MATLAB due to underlying parallel operations that are often opaque to the user. This can limit potential for improvement of serial code even for the so-called embarrassingly parallel applications. One such application is the computation of the Jacobian matrix inherent to most nonlinear equation solvers. Computation of this matrix represents the primary bottleneck in nonlinear solver speed such that commercial finite element (FE) and multi-body-dynamic (MBD) codes attempt to minimize computations. A timing study using MATLAB's Parallel Computing Toolbox was performed for numerical computation of the Jacobian. Several approaches for implementing parallel code were investigated while only the single program multiple data (spmd) method using composite objects provided positive results. Parallel code speedup is demonstrated but the goal of linear speedup through the addition of processors was not achieved due to PC architecture.

15. SUBJECT TERMS

MATLAB programming; Nonlinear equations; Parallel Jacobian

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
U	U	U	UU	19	19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658